Finite Programs, Infinite Digits: On the Algorithmic Determination of π and the Nature of Computable Reals

Douglas C. Youvan

doug@youvan.com

www.youvan.ai

November 9, 2025

The number π occupies a peculiar position at the intersection of geometry, analysis, and computability. Its decimal expansion is infinite and patternless in any obvious sense, yet it is also one of the best-understood constants in mathematics. This paper examines what it really means to "determine" π using only a finite algorithm, with no stored digits and no built-in π constant, and uses that question as a gateway into the broader theory of computable reals. From the standpoint of classical analysis, π is defined via limits, integrals, or geometric constructions. From the standpoint of computability, it is defined by the existence of a program that, given any desired precision n, returns an approximation to π with provably bounded error. We explore several classes of such algorithms—series expansions, polygonal schemes, and digit-extraction formulas—and analyze how each encodes the infinitude of π into finite symbolic rules. This leads naturally to an information-theoretic perspective: π as an infinite object with a surprisingly small Kolmogorov description. We contrast π with non-computable reals, which admit no such compression, and ask what this distinction implies about "discovery," "invention," and the structure of mathematical reality. In this way, π becomes a case study in how finite procedures can faithfully capture infinite truths.

Keywords: π, computable reals, algorithmic randomness, Kolmogorov complexity, infinite decimals, series expansions, digit extraction, Archimedean methods, information theory, mathematical philosophy.

A collaboration with GPT-5. 45 pages. CC4.0.

Outline

1. Introduction: What Does It Mean to "Determine" π?
   1.1 The many faces of π: geometric, analytic, and computational
   1.2 Infinite decimal expansion vs finite algorithmic description
   1.3 The distinction between "all digits" and "any finite prefix"

2. Computable Reals: Formal Definitions and Basic Notions
   2.1 Turing machines, programs, and effective procedures
   2.2 Definition of computable reals via effective approximation
   2.3 Error bounds, convergence rates, and constructive limits
   2.4 Countability of computable reals vs uncountability of R

3. Algorithmic Constructions of π Without Stored Constants
   3.1 Series-based methods
   3.1.1 Classical series (Leibniz, Machin-like formulas)
   3.1.2 Rapidly convergent series (e.g., Chudnovsky-type)
   3.1.3 Bounding truncation error for a given precision n
   3.2 Geometric and Archimedean methods
   3.2.1 Inscribed and circumscribed polygons around the unit circle
   3.2.2 Iterative doubling of sides and square-root computations
   3.2.3 Constructive root-finding as an algorithmic subroutine
   3.3 Digit-extraction and spigot algorithms
   3.3.1 Base-specific formulas (e.g., BBP-style methods)
   3.3.2 Local digit computation vs sequential digit generation
   3.3.3 Practical considerations: complexity and stability

4. Bootstrapping Without Stored Constants
   4.1 What "no stored constants" can reasonably forbid
   4.2 Building the naturals and rationals from minimal axioms
   4.3 Arithmetic, control flow, and the encoding of formulas
   4.4 Avoiding hidden π: libraries, "black box" constants, and purity

5. Kolmogorov Complexity and the Compression of π
   5.1 Programs as compressed descriptions of infinite sequences
   5.2 Descriptive complexity of π: small program, long expansion
   5.3 Comparison with random-looking but computable sequences
   5.4 π vs algorithmically random reals: structure and compressibility

6. Almost All Reals Are Non-Computable
   6.1 Cardinality argument: uncountably many reals, countably many programs
   6.2 Measure-theoretic intuition: computable reals as a “thin set”
   6.3 Examples and thought experiments for non-computable reals
   6.4 Implications for what “typical” real numbers look like

7. Philosophical Reflections on Determination and Knowledge
   7.1 “Knowing” π: finite algorithms vs infinite data streams
   7.2 Discovery vs invention: does π pre-exist its algorithms?
   7.3 Formalism, Platonism, and the status of computable structure
   7.4 Does algorithmic determinability capture mathematical essence?

8. Beyond π: Other Constants and Algorithmic Landscapes
   8.1 e, √2, and other classical computable constants
   8.2 Numbers defined by universal processes (e.g., halting probabilities)
   8.3 Constants at the boundary: definable but non-computable reals
   8.4 π as a pedagogical gateway into deeper computability phenomena

9. Conclusion
   9.1 Summary: π as a paradigmatic computable real
   9.2 Finite algorithms as windows onto infinite structures
   9.3 The epistemic meaning of “being able to compute π”
   9.4 Open questions: complexity, randomness, and the ontology of number

References

1. Introduction: What Does It Mean to "Determine" pi?

When people say that a computer "knows" pi, they often imagine some huge table of digits stored in memory or a magic constant hidden in a math library. From a computability perspective, that picture is exactly what we do *not* assume. Instead, we ask a sharper question: can a finite program, with no pre-stored digits of pi and no special pi-constant built in, systematically produce arbitrarily accurate approximations of pi on demand? If so, in what sense has pi been "determined" by that program?

This shifts attention from digits as static data to procedures as dynamic descriptions. The central contrast is between two impossible tasks and one possible one. No finite process can ever print "all" digits of an infinite decimal and then halt. Likewise, no finite formula can be expanded out into a literal, fully written-out decimal of pi. What *is* possible—and what defines pi as a computable real—is a procedure which, given a requested precision n, returns a rational number whose distance from pi is less than a prescribed error bound, for example 10^(-n). Determination, in this technical sense, means "effective approximability to any finite precision," not "completion of an infinite listing."

1.1 The many faces of pi: geometric, analytic, and computational

Historically, pi entered mathematics through geometry: it is the ratio of the circumference of any circle to its diameter. Archimedes approximated it using polygons; later work tied pi to areas of circles, volumes of spheres, and eventually to trigonometric and complex-analytic structures. Analytically, pi appears in integrals (e.g., the Gaussian integral), Fourier series, and the spectral behavior of many physical systems. Each of these classical definitions characterizes pi, but none of them by themselves tell us *how* to get its digits on an actual machine.

The computational face of pi starts when we recast these definitions as algorithms: explicit, finite recipes that a machine can follow. A geometric characterization becomes an iterative polygon scheme; an integral definition becomes a numerical integration routine; a series definition becomes a loop summing terms until an error bound is met. In this view, the essence of pi is not just that it is "circumference over diameter," but that there are concrete, executable methods that translate this relationship into a finite program capable of producing as much of pi's numerical content as we ask for.

Thus the same number pi can be seen as:

- a geometric ratio,
- an analytic constant woven through the fabric of calculus,
- and a computable real captured by a finite algorithm.

The last perspective is the one that makes sense of “determination” by a computer without stored constants.

1.2 Infinite decimal expansion vs finite algorithmic description

Every non-terminating, non-repeating decimal encodes an infinite amount of raw digit data. Naively, to “have” that number seems to require an infinite memory tape on which all its digits are written. But a finite program cannot literally contain infinitely many symbols. The only way a finite device can handle an infinite object is by compressing it into a rule: a description that can be unfolded step by step.

Pi’s decimal expansion is infinite, but the shortest program that generates it is finite. The program cannot finish writing all digits, but it can output the first n digits for any n, if you give it enough time. This is what makes pi computable. The infinite expansion is thus replaced by a *schema* for producing prefixes, together with proofs that the schema converges with a known rate and error bound.

Crucially, this finite description is not a mere shorthand. It is operational: it tells you exactly what to do to get approximations. In contrast, most real numbers cannot be encoded this way at all; they would require an infinite, incompressible list of digits with no shorter generating rule. For those numbers, “determination by algorithm” is impossible in principle. For pi, it is not only possible but routine. That gap between infinite expansion and finite description is where computability theory lives.

1.3 “All digits” vs “any finite prefix”: a necessary distinction

Informally, people sometimes say “we know all the digits of pi,” meaning that we know a formula or algorithm for generating them. Strictly speaking, no finite being or machine ever *possesses* all digits in material form; at any finite time, only a finite prefix has actually been written down or stored. What we really have is: for each finite n, a method that will eventually give us the first n digits, and a proof that the method always works.

This leads to a necessary distinction:

- All digits: an ideal, completed infinite object; unattainable by any finite physical process.
- Any finite prefix: a family of potential results, each of which can be realized by running the same algorithm longer or with a larger precision parameter.

In computability theory, the latter is sufficient to say “pi is determined by a program.” Determination here is *conditional* on how much you ask for, but *unconditional* in its guarantee that, for each finite request, the algorithm will succeed.

The philosophical tension is that our intuition often treats “knowing a number” as if a platonic, completed infinity is somehow grasped. The computational perspective replaces that with a hierarchy of effective procedures, each delivering finite, controllable approximations. The rest of the paper will build on this distinction, using pi as a running example to explore computable reals, algorithmic descriptions, and the thin boundary between what finite agents can and cannot determine about infinite mathematical objects.

2. Computable Reals: Formal Definitions and Basic Notions

If we want to say rigorously that “pi is computable,” we need more than intuition about “being able to approximate it.” We need a formal model of what an algorithm is, and a precise way to connect such algorithms to infinite objects like real numbers. This is the job of computability theory.

At a high level, a real number x is called computable if there exists a single, finite program that, when given a natural number n as input, outputs a rational number q(n) such that q(n) is within 2^-n of x. In other words, the program can “home in” on x with arbitrarily fine resolution, with the requested resolution encoded in n.

This section makes that picture precise. We start by recalling the classical notion of a Turing machine as an idealized model of computation, then use it to define computable reals via effective approximation. We will also emphasize the role of explicit error bounds and convergence rates in making limits constructive, and finally prove that there can be at most countably many computable reals, in stark contrast to the uncountable size of the real line R.

2.1 Turing machines, programs, and effective procedures

A Turing machine is a simple mathematical idealization of what it means to carry out a step-by-step procedure. Formally, it consists of:

- A finite set of internal states (including a start state and, optionally, one or more halting states),
- A bi-infinite tape divided into discrete cells, each containing a symbol from a finite alphabet,
- A read/write head that can move left or right along the tape,
- A finite transition table that, given the current state and the symbol under the head, specifies:
    - a symbol to write,

- a direction to move (left or right),
- and a new internal state.

A program in a modern language (C, Python, etc.) can be simulated by some Turing machine; conversely, the behavior of a Turing machine can be encoded as a program. The particular details do not matter for computability; what matters is that we have a precise, mechanical notion of an effective procedure: a finite rule set that, given an input, prescribes a unique sequence of discrete steps that either halts with an output or runs forever.

From this point of view, when we say “there exists an algorithm” we mean “there exists a Turing machine (equivalently, a program) with a specified input-output behavior.” The Church–Turing thesis says, informally, that this captures all reasonable notions of what a mechanical calculation can be.

We will write:

- f(n) is computable if there exists a Turing machine that, given input n coded on its tape, eventually halts and outputs f(n).

With this notion in hand, we can talk about sequences of rationals and the real numbers they converge to.

2.2 Definition of computable reals via effective approximation

Intuitively, a real number x is computable if there is a computable sequence of rationals that converges to x with a known, effective speed. The standard formalization is as follows.

A real x is computable if there exists a computable function q: N -> Q (from natural numbers to rational numbers) such that for every n in N,

abs(x - q(n)) <= 2^-n.

Here q(n) is the n-th approximation to x, and 2^-n is the required accuracy. The important thing is that:

- The map n -> q(n) is produced by a single finite program,
- The error bound is explicit and uniform: for each n, we know how close q(n) is to x.

Equivalently, we can define computable reals via binary expansions: x is computable if there is a Turing machine that, on input n, outputs the first n bits of x in binary. The two definitions are equivalent: from a fast-converging rational approximation one can extract correct initial digits, and from correct initial digits one can build rational approximations with small error.

Note how this fits our earlier intuition about pi. If we can implement a function q(n) that sums enough terms of some pi series to guarantee abs(pi - q(n)) <= 10^-n, then pi is computable. The critical point is that there is:

- A single algorithm that works for all n,
- A provable relationship between n and the approximation error.

The digit-by-digit view and the rational-approximation view are two faces of the same underlying idea: effective approximation.

2.3 Error bounds, convergence rates, and constructive limits

In classical analysis, we say a sequence (q(n)) converges to x if, for every epsilon > 0, there exists some N such that for all n >= N, abs(q(n) - x) < epsilon. This is a purely existential statement: for each epsilon there is some N, but there may be no effective way to find N from epsilon.

For computability, that is not enough. We want the convergence to be effective. There should be a computable function m: N -> N such that for all n:

abs(x - q(n)) <= 2^-n

or more generally, given k, we can effectively find N = m(k) such that for all n >= N, abs(q(n) - x) <= 2^-k.

Thus a computable real is not just a limit point in the usual sense; it is a limit for which the rate of convergence is controlled by a computable modulus. The error bounds and convergence rates are built into the algorithm.

This matters for pi. A naive infinite series may converge, but too slowly or in too opaque a way to be useful for computation. A practically useful pi algorithm is one in which, given the requested precision n, we can determine:

- How many terms to sum,
- How much rounding error is tolerable in intermediate steps,
- And how to guarantee that the final result is within the desired error bound.

In other words, the constructive limit viewpoint demands that limits be accompanied by explicit recipes for achieving specified accuracies. This is exactly what ties the analytic definitions of pi to the computational definition: a limit becomes computable when we can bound, in a computable way, how far we are from it after finitely many steps.

2.4 Countability of computable reals vs uncountability of R

Once we tie “computable” to “produced by a Turing machine,” a simple but profound counting argument appears.

1. Every Turing machine (or program) can be described by a finite string over a finite alphabet (its transition table or source code).
2. The set of all finite strings over a finite alphabet is countable: we can list them in order of length, and within each length in lexicographic order.
3. Therefore, the set of all Turing machines, and hence the set of all (partial or total) computable functions, is countable.

Now consider the mapping:

- program -> real number

where we send each program that computes a fast-converging rational approximation to the real x to that x. There may be many programs for the same real, but this does not affect cardinality: the image of a countable set under any mapping is at most countable.

Therefore:

- The set of computable reals is at most countable.

On the other hand, standard set theory tells us:

- The real line R is uncountable, for example by Cantor’s diagonal argument.

Combining these two facts, we get a strong conclusion:

- Almost all real numbers (in the sense of cardinality) are non-computable.

There is no program that, given n, can produce approximations to such a real with guaranteed error bounds. No algorithm can list its digits; no effective procedure can “determine” it in the sense we are using for pi.

This contrast is central to the philosophical point of the paper. Pi belongs to a very small, very special class of reals whose infinite structure can be compressed into a finite effective description. The existence of a pi-program is not typical of real numbers; it is exceptional. When we say that a computer can determine pi without stored constants, we are implicitly relying on this special compressibility. The next sections will explore how such programs are constructed and how much information about pi they really encode.

3. Algorithmic Constructions of pi Without Stored Constants

Once we know what it *means* for pi to be computable, the next question is: how do we actually build such algorithms without sneaking pi in through the back door?

The key is that pi appears as the value of many limits, sums, and geometric constructions that depend only on integers, rationals, and basic operations (addition, multiplication, division, square roots, etc.). Any of these can be turned into a program that, given a precision parameter n, produces an approximation to pi with guaranteed error.

This section surveys three major families of constructions:

1. Series expansions derived from analysis,
2. Geometric methods in the spirit of Archimedes,
3. Digit-extraction or “spigot” algorithms that operate at the level of individual digits.

All three can be expressed as finite rules operating on integers and rationals, with no pre-stored digits of pi.

3.1 Series-based methods

Series-based methods start from analytic identities where pi emerges as the value of an infinite sum or product. To get an algorithm, we turn “infinite” into “finite but good enough”: we truncate the series after some number of terms and bound the remainder.

The general pattern:

- Find a series pi = sum_{k=0}^\infty a_k where each a_k is a rational number (or at worst a rational combination of radicals).
- For given precision n, determine an N such that sum_{k > N} |a_k| <= 10^-n (or 2^-n).
- Compute partial sum S_N = sum_{k=0}^N a_k with enough internal precision that rounding error is controlled.

The algorithm is then:

- Input: n
- Compute N = N(n) using an explicit bound,
- Loop k = 0..N to accumulate S_N,
- Output S_N.

No pi is stored or assumed; it appears as the limit that this procedure converges toward.

3.1.1 Classical series (Leibniz, Machin-like formulas)

A famous example is the Leibniz series:

- pi/4 = 1 - 1/3 + 1/5 - 1/7 + ...

More generally:

- pi/4 = sum_{k=0}^\infty (-1)^k / (2k + 1).

This is beautiful, elementary, and completely constructible from integers, signs, and rational division. However, it converges extremely slowly; to get d correct decimal digits you need roughly 10^d terms. That is algorithmically valid but practically useless for large d.

A big step forward comes from Machin-like formulas, which express pi as a linear combination of arctan values at rational arguments, e.g.:

- pi/4 = 4 arctan(1/5) - arctan(1/239).

Since arctan(x) has a power series expansion:

- arctan(x) = sum_{k=0}^\infty (-1)^k x^(2k+1) / (2k+1),

we can write pi as a linear combination of series in powers of rational numbers like 1/5 and 1/239. These powers shrink quickly, so the series converges much faster than Leibniz's.

Algorithmically:

- We never store pi.
- We store small rational constants like 1/5, 1/239, and integer coefficients (4, 1).
- We implement the power series for arctan(x) as a loop.
- We control truncation based on the magnitude of the next term.

This is already a clean demonstration that pi can be computed purely from algebraic operations on integers and rationals.

3.1.2 Rapidly convergent series (e.g., Chudnovsky-type)

For serious computation, we use hyper-fast series, such as the Chudnovsky formula, which (in one common form) looks like:

- 1/pi = 12 * sum_{k=0}^\infty [(-1)^k (6k)! (13591409 + 545140134 k)] /
  [(3k)! (k!)^3 640320^(3k + 3/2)].

We do not need to memorize the exact coefficients here; the important features are:

- All coefficients are integers or rational powers of integers.
- Each term requires a finite combination of factorials, powers, and multiplications/divisions.
- The terms shrink extremely rapidly, giving about 14 extra correct digits per term.

The Chudnovsky formula, or similar Ramanujan-type series, are perfect for our theme:

- They encode an enormous amount of “digit information” about pi in a very compact finite expression.
- That expression can be turned into a loop using only integer arithmetic, rational arithmetic, and exponentiation with small integer exponents.
- Given a target precision n, we can determine how many terms are needed by bounding the magnitude of the next term in the series.

Thus, a short program implementing this formula and truncation rule is enough to generate millions or billions of digits of pi without ever storing a single digit of pi at the outset.

3.1.3 Bounding truncation error for a given precision n

Turning a series into a computable real requires controlling the tail:

- If pi = sum_{k=0}^\infty a_k, and S_N = sum_{k=0}^N a_k,
- Then the error is R_N = pi - S_N = sum_{k > N} a_k.

We need an explicit, computable upper bound on |R_N| in terms of N and the structure of a_k.

For many alternating series where |a_k| decreases monotonically, a simple bound holds:

- |R_N| <= |a_{N+1}|.

In such cases, to ensure |R_N| <= 10^-n, we just find N such that |a_{N+1}| <= 10^-n.

For more complicated series (like Chudnovsky), we can use ratio tests, bounding factorials and powers, to obtain inequalities of the form:

- |a_{k+1}| / |a_k| <= r(k),
- And then show that for k >= K(n) the terms are smaller than 10^-n.

The key point: for an algorithm to compute pi to n digits:

- It must be able to compute N(n) from n,

- And it must know that the tail beyond N(n) is smaller than the target error.

This is precisely the “effective approximation” condition from the computable-reals definition. The series plus its error analysis become a complete, finite specification of an algorithm for pi, independent of any stored value of pi.

3.2 Geometric and Archimedean methods

Long before infinite series, there was geometry. Archimedes’ classical method was to sandwich the circle between inscribed and circumscribed polygons and use their perimeters to bound pi.

Algorithmically, this is almost embarrassingly constructive: everything reduces to computing polygon side lengths via square roots and simple algebra.

3.2.1 Inscribed and circumscribed polygons around the unit circle

Consider a unit circle (radius = 1). For a regular n-gon:

- The perimeter of the inscribed n-gon, P_in(n), is less than 2 pi.
- The perimeter of the circumscribed n-gon, P_out(n), is greater than 2 pi.

Thus:

- P_in(n)/2 < pi < P_out(n)/2.

If we can compute P_in(n) and P_out(n) for larger and larger n, we get progressively tighter bounds on pi. All we need:

- The side length of the inscribed n-gon,
- The side length of the circumscribed n-gon.

Both can be expressed using trigonometric functions such as sin(pi/n) and tan(pi/n), but Archimedes effectively avoided trigonometry and instead used geometric relations that modern analysis interprets as involving square roots.

3.2.2 Iterative doubling of sides and square-root computations

A clever trick is to start with a hexagon (n = 6) and iteratively double the number of sides:

- From n-gon to 2n-gon to 4n-gon, etc.

There are recurrence relations that give the side length of the 2n-gon purely from the side length of the n-gon using square roots and rational operations. Schematically:

- If s_n is the side length of the inscribed n-gon in the unit circle, then s_{2n} can be computed from s_n via a formula involving sqrt(2 - sqrt(4 - s_n^2)), or similar.

All these formulas can be implemented with:

- Integer and rational arithmetic,
- Square root computations to required precision.

No pi appears explicitly; it is approached as the limit of polygon perimeters:

- pi = lim_{n -> infinity} P_in(n)/2 = lim_{n -> infinity} P_out(n)/2.

To turn this into an algorithm:

- Choose initial n (e.g., 6),
- Repeatedly apply the doubling formulas,
- After each step, compute lower and upper bounds for pi,
- Stop when the interval length (upper - lower) is smaller than 10^-n.

Crucially, the doubling process is iterative, and each step uses the previous step's result and a finite number of square-root operations.

3.2.3 Constructive root-finding as an algorithmic subroutine

Square roots are themselves limits: sqrt(a) is the value of x satisfying x^2 = a. To stay within a purely algorithmic world, we must also explain how to compute square roots without pre-stored constants.

This is straightforward:

- Use Newton's method (also called the Newton–Raphson method) for the function f(x) = x^2 - a.
- Start from a rational initial guess x_0 (e.g., x_0 = 1 or a rough rational bound),
- Iterate:

x_{k+1} = (x_k + a / x_k) / 2.

Under mild conditions, x_k converges rapidly to sqrt(a), and we can bound the error in terms of k. Again:

- The entire root-finding process is determined by a finite, explicit rule.

- The target precision n tells us how many iterations we need to get an approximation within 10^-n.

Thus, the geometric approximation of pi via polygons can be anchored entirely in:

- Integer/rational arithmetic,
- A rational iteration for square roots.

The circle, polygons, perimeters, and square roots together give a fully constructive framework for approximating pi without any built-in pi.

3.3 Digit-extraction and spigot algorithms

Series and polygons compute real-valued approximations to pi, from which we then extract digits. A different style of algorithm aims directly at the digits themselves.

A spigot algorithm produces digits one after another, like water drops from a spigot, without needing to store all previous digits at high precision, and sometimes even allowing one to jump to a specific position in the expansion.

3.3.1 Base-specific formulas (e.g., BBP-style methods)

One celebrated example is the Bailey–Borwein–Plouffe (BBP) formula, which expresses pi in base 16 (hexadecimal) as:

- pi = sum_{k=0}^\infty (1 / 16^k) * [4/(8k+1) - 2/(8k+4) - 1/(8k+5) - 1/(8k+6)].

What makes this remarkable is that:

- Using modular arithmetic and properties of base 16, one can compute the n-th hex digit of pi *directly*, without computing all preceding digits.
- This is still done purely by integer arithmetic, modular exponentiation, and rational operations—no stored digits of pi.

The BBP formula is base-specific: it is naturally expressed in base 16, and its digit-extraction trick works best there. But it proves an important conceptual point:

- Pi's digit sequence is not just computable sequentially; certain positions are computable locally from an explicit finite rule.

Again, the finite formula is serving as a compressed description of an infinite digit stream.

3.3.2 Local digit computation vs sequential digit generation

We can distinguish two kinds of digit algorithms:

1. Sequential:
    - Produce digits d_1, d_2, ..., d_n in order.
    - To get digit d_n you typically have to compute and maintain all earlier ones (or at least some high-precision rational approximations that encode them).
2. Local / random-access:
    - Given n, compute d_n directly (or with only polylogarithmic overhead in n),
    - Without needing all previous digits.

Classical series and geometric methods are naturally sequential digit generators: you refine a rational approximation and then peel off more digits.

BBP-style formulas and related constructions are closer to random-access: the n-th digit is computed from sums that involve n directly, with modular arithmetic designed so that earlier digits are not needed.

Both approaches fit into the computable-reals framework: they provide an effective procedure to get any desired prefix. But they differ in how the finite description “indexes” into the infinite digit sequence.

From the perspective of “determining pi without constants,” either style counts. The underlying requirement is that there exists a program that, for arbitrary n, outputs digit d_n, or the first n digits in sequence.

3.3.3 Practical considerations: complexity and stability

From a purely theoretical standpoint, any of the above constructions establishes pi’s computability. In practice, however, we care about:

- Time complexity: How does runtime grow with n, the number of digits we want?
    - Slow series like Leibniz are O(10^n) terms per n digits: hopeless for large n.
    - Chudnovsky-type series have time roughly polynomial in n and are used in real-world record computations.
- Space complexity: How much memory is required?
    - High-precision arithmetic requires storing big integers with O(n) bits.

    - Clever algorithms and number representations reduce overhead.
- Numerical stability: How sensitive is the method to rounding errors?
    - Some series are well-conditioned; truncation and rounding errors are easy to bound.
    - Others may involve cancellation or ill-conditioned steps requiring higher internal precision.
- Implementation details:
    - Efficient big-integer libraries, FFT-based multiplication, and modular algorithms all matter.

Yet all of these engineering concerns are *secondary* to the main conceptual point of the paper: that the essence of "determining pi" algorithmically lies in the existence of a finite, explicit rule that produces arbitrarily accurate approximations or digits, without any stored numeric representation of pi itself.

Series, geometric methods, and digit-extraction formulas are different ways of encoding the same infinite object—pi—into finite symbolic procedures. They differ in convergence speed and implementation complexity, but philosophically they all demonstrate the same fact: pi's infinite structure is fully captured by a finite algorithmic description.

4. Bootstrapping Without Stored Constants

So far we've treated "no stored constants" in an intuitive way—no π in memory, no lookup table of digits. But if we push on this phrase, it quickly becomes slippery. A program *must* contain some constants: at minimum, symbols, states, and the finite text of the source code itself. The real issue is not the *existence* of constants, but which ones we allow and what we count as "cheating."

This section clarifies what "no stored constants" can reasonably forbid, and then shows how we can still build everything we need—natural numbers, rationals, arithmetic—on top of a very small base. We also consider the role of libraries and "black box" constants, and why a purity requirement matters if we want to claim that π has been derived rather than smuggled in.

4.1 What “no stored constants” can reasonably forbid

If we interpret “no stored constants” too literally, we get an impossible requirement:

- A program with *zero* constants would have no characters, no instructions, no states—no content at all.

That’s clearly not what we mean. Instead, we are targeting a more specific prohibition:

- No pre-encoded numerical value of π (no digits, no closed-form rational approximation, no built-in “pi” symbol that the runtime interprets as a numerical constant).
- No indirect π by design (no obscure constant c chosen such that c = 10^1000 * π truncated, etc.).

Reasonable constraints, then:

1. Primitive symbols are allowed.
   We must allow basic lexical constants: symbols like 0, 1, +, -, *, /, parentheses, variable names, and the program structure itself.
2. Finite rational constants are allowed.
   Integers and small rationals such as 2, 3, 1/5, 1/239 are fine. They can be understood as part of the definition of the algorithm, not as precomputed knowledge of π. Many analytic formulas for π use such rationals.
3. Structural constants are allowed.
   Loop bounds, coefficients in series, offsets in formulas, and so on—provided they do not themselves encode π in disguise.

What is forbidden is essentially *semantic π*: any constant that, if interpreted numerically, would already be a high-precision approximation to π. If we avoid that, then we can legitimately say that π appears as an emergent value of computation, not as something we quietly loaded from memory.

4.2 Building the naturals and rationals from minimal axioms

Suppose we want to go all the way down. Can we build the numbers we need from almost nothing?

At a logical level, we can base everything on:

- A single distinguished object (often called 0),

- A successor operation S that maps each number to the next one,
- A collection of axioms (e.g., Peano axioms) that characterize the natural numbers.

In programming terms, we might conceptually:

- Represent 0 by some canonical bit pattern or state.
- Define S as “append a tick,” or “increment an integer,” or “go to the next configuration.”

From these, we can construct:

- Naturals N: 0, S(0), S(S(0)), …
- Integers Z: by introducing a sign bit or representing them as differences of naturals.
- Rationals Q: as pairs of integers (p, q) with $q \neq 0$, subject to an equivalence relation (p1, q1) ~ (p2, q2) if p1 q2 = p2 q1.

Practically, a program doesn’t need to *explicitly* implement Peano arithmetic from raw tape symbols to be legitimate. But conceptually, it’s important that:

- The numerical universe used by the algorithm (integers, rationals) can be grounded in a finite symbolic system and simple constructors.
- All of this is independent of π; π only appears when we start applying series, limits, or geometric constructions on top of this base.

So even under a strict reading, π is not built into the number system itself. The number system is generic; π is a particular real that emerges from specific constructions within that system.

4.3 Arithmetic, control flow, and the encoding of formulas

Given basic numbers, we need three things to get actual π algorithms:

1. Arithmetic operations
    - Addition, subtraction, multiplication, and division on integers and rationals.
    - Possibly exponentiation by small integers, factorials, and square roots via iterative methods.
    - These can all be defined in terms of repeated addition, loops, and simple combinators. Their correctness does not depend on π.

2. Control flow
    - Conditionals (if ... then ... else ...), loops (for, while), and recursion.
    - These allow us to encode finite procedures like “sum the first N terms” or “iterate Newton’s method until the error is below 2^-n.”
3. Encoding of formulas
    - A series like π/4 = ∑ (-1)^k / (2k+1) can be written as pseudocode:
        - set sum = 0,
        - for k = 0 to N, add (-1)^k / (2k+1) to sum,
        - output 4 * sum.
    - The formula’s structure—its coefficients, exponents, and denominators—is stored symbolically as part of the program, not as numerical approximations of π.

This distinction is crucial:

- The program text encodes a *finite rule* (“sum this series”),
- The run-time behavior yields *numeric approximations* to π,
- The constants in the code (like 4, 2k+1, etc.) are not approximations to π; they are structural parameters of the rule.

From the viewpoint of computability, a formula for π is legitimate if:

- Every symbol and constant in it is independently definable (e.g., small integers and rationals),
- The procedure can be implemented purely via arithmetic and control flow.

No step in the computation requires reading a π-digit from memory; all digits appear as a consequence of repeated, local operations dictated by the formula.

4.4 Avoiding hidden π: libraries, “black box” constants, and purity

In real software, we rarely work at this austere level. We write:

- double pi = 3.141592653589793;
- or double pi = acos(-1.0);

- or we call M_PI from a math library.

All of these hide π in some way:

- A literal decimal constant is obvious smuggling.
- The acos(-1.0) trick is more subtle: you are leveraging the library's implementation of trigonometric functions, which itself encodes knowledge about π through its internal approximations to cosine or its use of identities like cos(π) = -1.

From a *practical* engineering standpoint, this is fine. From the standpoint of this paper, however, such usage:

- Delegates the core work of determining π to an opaque, pre-existing module,
- Violates the purity requirement that π should be produced from scratch by a transparent algorithm.

We can formalize a purity condition:

- An algorithm for π is pure if every numeric constant in the program is either:
    - A small integer or rational chosen independently of π, or
    - A bound or parameter derived from earlier steps of the same algorithm,
- And if it uses no external routines whose behavior depends on precomputed knowledge of π.

Under this condition:

- A Chudnovsky-series implementation where all coefficients are spelled out in the source is pure.
- A polygon-doubling method using Newton's method for square roots is pure, provided Newton's method is implemented explicitly and not via a black box sqrt() that might hide a π-based implementation of trig.

Of course, in reality, sqrt() rarely hides π, but the conceptual point is: we want the entire path from integers to π to be inspectable and justified without appeal to a hidden constant.

In summary:

- "No stored constants" does not mean "no constants at all." It means "no pre-stored *answer*."

- A legitimate π algorithm bootstraps from minimal numerical primitives, uses arithmetic and control flow to implement analytic or geometric constructions, and keeps all uses of external code transparent with respect to π.
- When those conditions are met, we can credibly claim that the program determines π algorithmically, rather than simply retrieving π from memory under a different name.

5. Kolmogorov Complexity and the Compression of π

We have argued that a finite program can determine π to arbitrary precision without any stored digits. Kolmogorov complexity gives us a way to make that intuition precise: it treats programs as *compressed descriptions* of objects—here, infinite digit sequences. From this viewpoint, π is not just computable; it is *highly compressible*. A short algorithm plus a small amount of "overhead" can generate as many digits as we wish.

This section introduces Kolmogorov complexity as a formal language for talking about compression, then uses it to contrast π with both pseudorandom-looking but computable sequences and truly algorithmically random reals. The punchline is subtle but important: π's digits can look random while the number itself remains extremely low in descriptive complexity.

5.1 Programs as compressed descriptions of infinite sequences

Kolmogorov complexity starts from a simple idea:

- The Kolmogorov complexity $K(s)$of a finite string $s$is the length (in bits) of the *shortest program* that outputs $s$and halts (for some fixed universal programming language or universal Turing machine).

Intuitively:

- If $s$has a lot of internal structure or regularity, then a short program can generate it, so $K(s)$is small relative to | s |.
- If $s$looks patternless, then the best you can do is "hard-code" it, so $K(s)$is roughly | s |: there is no shorter description than the string itself.

A few important points:

- The exact value of $K(s)$is uncomputable in general; we use it conceptually, not as a practical function.

- Different programming languages or universal machines change $K(s)$by at most an additive constant, so the notion of "more or less compressible" is robust.

For an infinite sequence $x = x_1x_2x_3$ …, we look at its finite prefixes $x \upharpoonright n = x_1x_2 \dots x_n$. The growth rate of $K(x \upharpoonright n)$as a function of $n$tells us how compressible the sequence is:

- If there exists a constant $c$such that $K(x \upharpoonright n) \leq c + O(\log\ n)$for all n, then the sequence is extremely compressible: one fixed algorithm plus an encoding of $n$suffices to generate the first $n$symbols.
- If typically $K(x \upharpoonright n) \approx n$, then the sequence behaves like an incompressible random string.

In this sense, a program is a compressed description of the infinite sequence it generates. The shorter the program relative to the length of the sequence's prefixes, the higher the compression.

5.2 Descriptive complexity of π: small program, long expansion

Now consider the infinite decimal (or binary) expansion of π. For each n, let $\pi \upharpoonright n$be the first $n$digits of π in some fixed base.

Because π is computable, there exists:

- A single finite program Pthat, given n, outputs $\pi \upharpoonright n$and halts.

We can turn this into an upper bound on Kolmogorov complexity:

- To describe $\pi \upharpoonright n$, one way is to:
    - Write down the code of P,
    - Hard-code the value of $n$into a small input-reading stub,
    - Run Pon that hard-coded input.

The length of this description is roughly:

- $|\ P\ |$ +length of the encoding of n.

Encoding $n$in binary costs about $\log_2 n$bits. So we get:

- $K(\pi \upharpoonright n) \leq c + O(\log\ n)$,

where $c$is a constant capturing the size of the π-algorithm itself (e.g., the Chudnovsky series implementation, polygon algorithm, or BBP formula plus the needed control logic).

This is an *extreme* level of compression:

- The prefix $\pi \upharpoonright n$has length n,
- But its shortest description needs only on the order of $\log n$bits more than a fixed constant.

In other words:

- A small program + a brief encoding of how many digits you want suffices to reconstruct an arbitrarily long prefix of π.
- The descriptive complexity grows *logarithmically*, while the string itself grows *linearly*.

That is what it means, in Kolmogorov terms, to say that π is *highly compressible* despite its infinite, non-repeating expansion.

5.3 Comparison with random-looking but computable sequences

The digits of π are empirically very "random-looking":

- They appear uniformly distributed in base 10 and base 16 (at least up to the trillions of digits we have checked),
- They pass many statistical tests for randomness.

But randomness in the statistical sense is not the same as randomness in the algorithmic sense of Kolmogorov complexity.

Many sequences share this dual nature:

- Champernowne's constant in base 10, for example, is constructed by concatenating the decimal representations of the natural numbers:

$C_{10} = 0.12345678910111213\ldots$

It is known to be *normal* in base 10 (digits are uniformly distributed in the limit), so its digits look statistically random. Yet it has a trivial description:

"Write the natural numbers in order and stick them together."

The corresponding program is tiny, so $K(C_{10} \upharpoonright n) \leq c + O(\log\ n)$as with π.

- Pseudorandom number generator outputs are similar. The seed and the generator algorithm form a very short description, yet the output stream can pass many randomness tests.

π fits into this category: a random-looking but computable sequence. Its digits are conjectured to be normal (though not yet proved in base 10), and the early sampled digits exhibit no obvious patterns. Still, the existence of a short π-program forces its prefixes to be highly compressible in the Kolmogorov sense.

So we get a two-level picture:

- At the surface level, π's digits behave statistically like a "high-entropy" sequence.
- At the description level, π is overwhelmingly low-entropy: a short finite rule generates the whole structure.

This distinction is key when we compare π with truly algorithmically random reals.

5.4 π vs algorithmically random reals: structure and compressibility

An infinite sequence $x$is algorithmically random (in the sense of Martin-Löf randomness) if there is no effective statistical test that distinguishes $x$from a sequence generated by independent fair coin flips. A deep theorem connects this to Kolmogorov complexity:

- For an algorithmically random sequence x, there exists a constant $c$such that for all n:

$$K(x \upharpoonright n) \geq n - c.$$

So for "truly random" reals:

- Every prefix is essentially incompressible,
- The best description is almost just "print this prefix directly."

Moreover:

- Algorithmically random reals are uncomputable.
- If a sequence were both computable and algorithmically random, then the program generating it would compress its prefixes, contradicting the incompressibility condition.

By contrast, π is:

- Computable: there exists a finite algorithm to generate its digits.
- Low in Kolmogorov complexity: its prefixes have $K(\pi \upharpoonright n) \leq c + O(\log\ n)$.

Therefore, regardless of how random the digits of π may *appear*, π cannot be algorithmically random in this strict sense. Its existence as a computable constant places it in a small, structurally rich subset of the reals.

This yields an important conceptual contrast:

- Algorithmically random reals (like Chaitin's Ω numbers) embody *maximal* information: no finite rule compresses them; they cannot be determined by any algorithm.
- π embodies *structured* information: its infinite digit sequence is the unfoldment of a very short finite rule.

From the perspective of this paper, that is the heart of the matter:

- The fact that a finite program can determine π to any precision is equivalent to saying that π's infinite structure is completely encoded in a compact algorithmic description.
- "No stored constants" does not mean "no information about π"; it means that the information is carried not as digits, but as structure—as relations, limits, and identities that a machine can execute.

In Kolmogorov terms, the program is the compressed form; the infinite decimal expansion is the decompressed output.

6. Almost All Reals Are Non-Computable

So far we've focused on π as a *paradigm* of computability: a real number whose infinite structure is fully encoded in a finite program. It is natural—almost seductive—to assume that "numbers are like this in general." The opposite is true. In the space of all real numbers, π and its computable cousins form an exceedingly rare, thin dusting. Almost every real number, in a precise sense, is fundamentally beyond algorithmic reach: there is no program that can generate its digits with guaranteed accuracy.

This section makes that claim precise. First, via a cardinality argument: there are only countably many programs, but uncountably many reals. Then via measure-theoretic intuition: the set of computable reals has measure zero—like a dusting of points in an otherwise continuous interval. We then consider examples and thought experiments that make non-computable reals more concrete, and finally reflect on what this means for our intuitive picture of "typical" numbers.

6.1 Cardinality argument: uncountably many reals, countably many programs

The foundational argument is almost embarrassingly simple, yet its consequences are profound.

1. Programs are finite strings.
   Any program (or Turing machine description) can be written as a finite string over a finite alphabet: ASCII characters, machine opcodes, whatever.

2. Finite strings over a finite alphabet are countable.
   We can enumerate all such strings systematically:

   - First list all 1-character strings in lexicographic order,
   - Then all 2-character strings,
   - Then 3-character strings, and so on.

This gives a sequence that, in principle, includes every possible program text.

3. So the set of programs is countable.
   Some of these strings are nonsense; some are well-formed but non-terminating; some compute functions N → N, some N → Q, and some N → prefixes of a real. But as a *set*, there are only countably many distinct programs.

Now, each computable real number is determined by at least one program that delivers its approximations (or its digits). Even if many different programs compute the same real, the total collection of distinct computable reals is no larger than the set of all programs. Therefore:

- The set of computable reals is at most countable.

By contrast, classical set theory tells us:

- The set of all real numbers R is uncountable, as shown by Cantor's diagonal argument. No enumeration r_1, r_2, r_3, … can capture them all; we can always construct a real that differs from each r_n in its n-th digit.

Putting this together:

- The computable reals form a set that is, at most, countable.
- The full continuum of reals is uncountable.

So "most" real numbers (in the sense of cardinality) cannot possibly be computable: there just are not enough programs to go around. For every computable real, there are uncountably many others with no algorithmic description at all.

6.2 Measure-theoretic intuition: computable reals as a “thin set”

Cardinality tells us that non-computable reals vastly outnumber computable ones, but the notion of “almost all” can be sharpened using measure theory.

Consider the real interval [0, 1]. Each real in this interval corresponds to an infinite binary expansion:

- $x = 0.b_1b_2b_3\ldots$

with $b_k \in \{0, 1\}$. The Lebesgue measure on [0, 1] is what we usually mean by “length” or “probability density” on this interval.

Key facts:

1. Any countable set has Lebesgue measure zero.
    - You can cover each point by a tiny interval of length ε/2, ε/4, ε/8, …
    - The total length of all these intervals is at most ε.
    - Since ε is arbitrary, the set can be covered by intervals of arbitrarily small total length.
2. Since the computable reals are at most countable, they have measure zero in [0, 1].

In probabilistic language:

- If you pick a real number in [0, 1] “at random” according to the uniform distribution, the probability that you pick a computable real is 0.
- Yet computable reals certainly exist (π, e, √2, etc.); they are just “invisible” to measure, like a dust of points in a continuous fog.

This is the “thin set” intuition:

- The computable reals are dense: between any two reals, there is a computable one.
- But they are also measure-zero: they occupy no “volume” in the continuum.

So we get a curious duality: computationally accessible numbers are dense enough to approximate anything, yet so sparse that a randomly chosen real will almost surely be forever outside algorithmic reach.

6.3 Examples and thought experiments for non-computable reals

Non-computable reals can feel abstract, even mystical—numbers that "exist" but cannot be fully grasped by any program. But there are explicit constructions that force non-computability.

1. The characteristic real of the Halting Problem.
   Enumerate all Turing machines M_1, M_2, M_3, … in some canonical order. Define an infinite binary sequence h:
   - $h_n = 1$ if machine M_n halts on empty input,
   - $h_n = 0$ if it does not.

Interpret h as a binary expansion:

- $H = 0.h_1h_2h_3…$

If H were computable, there would be a program that, given n, outputs h_n. But computing h_n means deciding whether M_n halts, which would solve the Halting Problem—known to be undecidable. Thus:

- H is a well-defined real number but non-computable.

2. Chaitin's Ω numbers.
   Fix a universal prefix-free Turing machine U. Define:
   - Ω = sum over all halting programs p of 2^{-|p|}

Intuitively, Ω is the "probability" that a random program halts. Ω is a real number in (0, 1) whose binary expansion encodes the halting behavior of all programs. One can show:

- Ω is not only non-computable, but algorithmically random: its prefixes have maximal Kolmogorov complexity.

So Ω is a concrete, mathematically defined real that is provably beyond algorithmic determination.

3. Diagonalization against all computable reals.
   Suppose $\{x_1, x_2, x_3, …\}$ is a computable enumeration of all computable reals in [0, 1], each with binary expansion $x_n = 0.x_{n1}x_{n2}x_{n3}…$. Define a new sequence y by:
   - $y_n = 1$ if $x_{nn} = 0$, and $y_n = 0$ if $x_{nn} = 1$.

Then $y = 0.y_1y_2y_3…$ differs from each x_n in its n-th bit, so y cannot be equal to any computable real in the list. If the enumeration were actually complete, y would be a concrete description of a non-computable real.

These examples show that non-computable reals are not vague metaphysical ghosts: they arise naturally whenever we encode undecidable problems (like the Halting Problem) into the digits of a number.

A useful thought experiment:

- Imagine flipping a fair coin infinitely many times to generate a binary expansion $0.b_1b_2b_3\ldots$
- With probability 1, the resulting real is algorithmically random and thus non-computable.
- There is no finite program that can predict its bits beyond trivial chance.

So "non-computable" is not rare or exotic; it is the default behavior of a random real.

6.4 Implications for what "typical" real numbers look like

Putting all this together, we can now contrast the numbers we *care about* (like π) with the numbers that are *typical* in a mathematical sense.

- Numbers we use in mathematics and physics
    - π, e, √2, algebraic numbers, well-known constants defined by series, integrals, or differential equations.
    - All of these are highly structured and (as far as we know) computable.
    - Their infinite behavior is captured by simple laws, formulas, or algorithms.
- Typical reals in R
    - Almost all are non-computable.
    - Their digit sequences are algorithmically random: no finite pattern, no finite rule, no compressible structure encodes them.
    - They cannot be "determined" by any machine in the sense we've defined for π.

This leads to a striking inversion of intuition:

- From the human perspective, π is "complicated": we learn special formulas, we build supercomputers to compute its digits, we write papers about its properties.
- From the perspective of the entire continuum, π is exceptionally simple: a tiny algorithmic island of structure in an ocean of uncomputable, incompressible numbers.

For the theme of this paper—"Can π be determined algorithmically without stored constants?"—the implication is:

- The very *fact* that such a question makes sense already places π in a privileged class.
- The existence of any finite program that succeeds, no matter how slow or cumbersome, distinguishes π from almost every real number one could conceivably name or "choose at random."

In other words, when we show that a constant like π is computable, we are not making a trivial observation about "how numbers work." We are identifying π as part of a rare and highly structured subset of the reals whose infinite content is fully compressible into a finite algorithm. The rest of the continuum—"typical" reals—will never enjoy such a determination, no matter how clever the program or how long we are willing to wait.

7. Philosophical Reflections on Determination and Knowledge

By this point, the technical story is reasonably clear: π is computable, its digits can be generated by a finite program without any stored π-constant, and this distinguishes it sharply from the vast majority of real numbers. What remains is the philosophical sting in the tail: *what does that actually tell us about "knowing" π, about its existence, and about the nature of mathematics itself?* This section treats π as a test case for broader questions about determination, discovery, and the status of computable structure.

7.1 "Knowing" π: finite algorithms vs infinite data streams

There are at least two very different ways to think about "knowing" a real number like π:

1. Data-stream picture
    - To know π is to "have" its entire decimal (or binary) expansion: an infinite list of digits extending without end.
    - On this view, our computational situation is always partial and provisional; we only ever know some initial segment of π's expansion, never the whole thing.
2. Algorithmic picture
    - To know π is to possess an *effective method* that can, for any given precision n, produce an approximation accurate to 10^(-n) (or 2^(-n)).

- Here, the focus is not on possessing a completed infinity of digits but on being able to generate any finite prefix on demand.

Our analysis in previous sections squarely favors the algorithmic picture. A finite program encodes the "how" of π: how to approach it, how to approximate it, how to refine our knowledge as far as we wish. From this point of view, a mathematical agent—human or machine—"knows" π when it has internalized a correct algorithm for its approximation, plus a proof that the algorithm converges as claimed.

That does not dissolve the allure of the data-stream picture, especially if one leans toward a more Platonic view of mathematics. A Platonist might say: the algorithm gives us *access* to π, but π itself is an infinite object that exists independently of our procedures. The finite program is a lens; the infinite sequence is what the lens reveals. The tension between those pictures—knowledge as possession vs knowledge as generative capacity—runs through much of modern philosophy of mathematics.

7.2 Discovery vs invention: does π pre-exist its algorithms?

The question "Is π discovered or invented?" is often treated as a slogan, but in the context of computability it becomes more concrete.

Consider that π can be specified in many ways:

- As the ratio of circumference to diameter in Euclidean geometry,
- As twice the smallest positive root of sin(x) = 0,
- As the sum of a Chudnovsky-type series,
- As the limit of Archimedean polygon procedures,
- As the output of a specific Turing machine computing its digits.

All of these different formulations converge—literally and conceptually—on the same real number. When programmers write a new π-algorithm, they are not changing π; they are adding another bridge from our symbolic systems to that same, stable target.

A Platonist can read this as strong evidence for discovery:

- π is a pre-existing object in a mathematical universe of truths.
- Our various geometric, analytic, and algorithmic characterizations are different ways of *finding* it.

- The fact that independent lines of reasoning converge on the same constant suggests that π has an identity not reducible to any single human-made formalism.

A formalistic or constructivist viewpoint may lean differently:

- π is not a mysterious entity floating in a separate realm; it is the *equivalence class of all its correct procedures and theorems*.
- To "be" π is just to be the unique real satisfying certain definitional conditions inside our axiomatic systems.
- On this view, our programs and proofs are not windows into a separate world, but the *fabric* of what π means.

Computability theory does not force a resolution, but it sharpens the terms of debate: whatever π "really" is, the fact that it can be captured by short, finite algorithms suggests that its essence is tightly bound up with structure, invariance, and cross-linked definitions—far more than with any particular representation or data store.

7.3 Formalism, Platonism, and the status of computable structure

Computable reals sit in a sweet spot between these philosophies.

From a formalist standpoint:

- Computable reals are the ones that can be fully "lived inside" a symbolic system.
- They are determined by axioms plus effective rules of inference and calculation.
- The fact that π is computable means that everything about it that we can need, in practice, can be instantiated inside formal procedures: approximations, inequalities, series expansions, error bounds.

From a Platonist standpoint:

- Computable reals are the ones our finite minds and machines can *track* through the infinite mathematical landscape.
- They occupy a privileged position not because they are the only reals that exist, but because they are the only ones whose full structure can be unfolded, step-by-step, by finite agents.

Both views can agree on a key point:

- Computable structure marks the boundary of effective mathematical knowledge.

- If a real is non-computable, no amount of clever programming or resource scaling will ever produce its digits in a systematic, guaranteed way.

π is therefore part of a "visible" mathematical frontier that both formalists and Platonists can endorse, even if they disagree about what lies beyond. For formalists, the frontier is the whole story; for Platonists, it is the illuminated edge of a much larger, mostly inaccessible terrain.

7.4 Does algorithmic determinability capture mathematical essence?

One might be tempted to say: if a number is not computable, it is somehow less real or less mathematically meaningful. After all, we cannot algorithmically determine its digits; we cannot encode it in a program; we cannot do with it what we do so comfortably with π.

But this temptation runs into trouble quickly:

- Many standard mathematical objects are defined in ways that (provably) produce non-computable reals—e.g., the Ω constants associated with halting probabilities, or reals defined by transfinite constructions, or by diagonalization against all algorithms.
- These objects play genuine roles in proofs about what can and cannot be computed, about the limits of formal systems, and about randomness.

So algorithmic determinability captures one kind of mathematical essence: the essence of *effective structure*, the part of mathematics that can be unfolded by finite procedures. It does not capture all mathematically significant objects.

π is emblematic of the computable side:

- Its identity can be completely encoded in finite axioms plus effective inference rules.
- Its digits can be generated by transparent programs, with proofs of correctness.
- In that sense, π is a model citizen of the algorithmic universe: everything we want to say about it can, in principle, be translated into program behavior and provable termination.

Non-computable reals, by contrast, embody a different kind of essence:

- They are where mathematics touches the limits of method.
- They witness that not all well-defined properties or sets of integers can be packaged into finite algorithms.
- They show that the landscape of "true but uncomputable" phenomena is vast.

Thus, the answer to the overarching question—"Does algorithmic determinability capture mathematical essence?"—is both yes and no:

- Yes, for the part of mathematics that concerns what finite agents can systematically, provably, and constructively access. π lives here, and its computability is a key part of its identity.
- No, if we take "mathematical essence" to include all that is definable or logically coherent within our theories, including objects whose full content will forever exceed computational reach.

Seen in this light, examining π's computability without stored constants is not just a technical exercise. It is a way of clarifying the boundary between what can be *generated* by finite means and what must remain, in some sense, merely *indicated* by our formal systems. π sits right on that boundary: an infinite object, yet one whose entire infinite unfolding is coded into a finite algorithm—a bridge between the finite mind and the infinite structure it seeks to understand.

8. Beyond π: Other Constants and Algorithmic Landscapes

π is a friendly door into the world of computable reals: familiar, geometric, and clearly definable. But it is not unique in this regard. Many other important constants are just as computable as π, while others live at the far edge of what can be defined or computed. Stepping beyond π lets us see that "algorithmic determination" is not a quirk of one special number, but part of a broader landscape of relationships between structure, definability, and computability.

This section sketches that landscape: first by looking at other classical computable constants; then by examining numbers defined by universal computational processes, such as halting probabilities; then by touching the boundary where definability outruns computability; and finally by returning to π as a pedagogical gateway to these deeper phenomena.

8.1 e, √2, and other classical computable constants

Everything we have said about π generalizes smoothly to many familiar constants:

- e, the base of natural logarithms, can be defined analytically by
  e = lim_{n→∞} (1 + 1/n)^n
  or by the series
  e = ∑_{k=0}^∞ 1/k!.
  Both definitions give directly computable algorithms: for any n, we can sum enough

terms or pick a large enough n in the limit to approximate e within 10^(-n), with explicit error bounds.

- √2, the length of the diagonal of a unit square, is computable via:
    - Iterative methods (e.g., Newton's method for x^2 - 2 = 0),
    - Binary search between rational bounds (e.g., between 1 and 2),
    - Continued fractions (√2 has a simple, periodic continued fraction expansion). All of these are finite procedures operating on rationals, converging to √2.
- Other algebraic numbers (roots of polynomial equations with integer coefficients) are generally computable: root-finding algorithms (for polynomials) can isolate and approximate them to arbitrary precision.
- Many transcendental constants—e^2, π^2, ln(2), ζ(3), etc.—are similarly computable because they arise as values of functions that themselves can be approximated algorithmically (series, integrals, functional equations, etc.).

What unites all these examples is that:

- They are defined by simple, effective laws (polynomial equations, convergent series, limits of explicit sequences),
- These laws can be turned into programs that approximate the constants as closely as desired.

π is in this sense just one member of a large and densely interconnected family of computable constants that underwrite analysis, geometry, and mathematical physics.

8.2 Numbers defined by universal processes (e.g., halting probabilities)

Things get more interesting when we consider constants defined not by simple formulas, but by *universal computational behavior*. The prime example is Chaitin's Ω:

- Fix a universal prefix-free Turing machine U.
- Define Ω_U as the probability that U halts when fed a random program (where programs are random bitstrings and the prefix-free condition ensures a proper probability distribution).

Formally:

- Ω_U = ∑_{p halts} 2^{-|p|}.

This is a real number in (0, 1) whose binary expansion encodes, in a compressed way, the halting behavior of *all possible programs* for U.

Key facts:

- Ω_U is well-defined, but non-computable. No algorithm can approximate Ω_U to arbitrary precision: knowing enough bits of Ω_U would solve infinitely many instances of the Halting Problem.
- The bits of Ω_U are algorithmically random: there is no shorter description of its n-bit prefixes than the prefixes themselves (up to a fixed additive constant).

Other universal or near-universal constants include:

- Reals encoding the Busy Beaver function values (maximal runtime of n-state Turing machines that eventually halt),
- Reals that encode, in their digits, answers to all provable statements in a formal theory,
- Probabilities in random process models whose behavior is tied to undecidable properties.

These constants illustrate how far we can go with "number as summary of process." Whereas π summarizes geometric and analytic regularities, Ω summarizes the computational behavior of an entire universe of programs. But the cost of such universality is non-computability: the corresponding numbers cannot be algorithmically determined in the way π can.

8.3 Constants at the boundary: definable but non-computable reals

Between "nice computable constants" like π and e and "wild" algorithmically random reals, there is a large frontier of numbers that are definable in precise mathematical terms but still non-computable.

Examples and types:

- Diagonal reals over computable reals.
  Given a computable list of all computable reals (under some encoding), we can construct a new real by diagonalization (changing the n-th digit of the n-th real). This new real is:
    - Precisely defined from the list,
    - Guaranteed not to be in the list,
    - Therefore non-computable.

- Reals encoding undecidable sets.
  Take any non-computable set of natural numbers S (e.g., the Halting set). Form a real:
    - x = ∑_{n in S} 2^(-n)
      or interpret the characteristic function of S as a binary expansion.
      Then x is non-computable but perfectly well-defined once S is defined.
- Reals defined by logical strength.
  There are reals whose definability depends on strong set-theoretic axioms or large cardinal hypotheses, and which are non-computable from weaker systems. Their status is tied to the logical strength of the theories we accept rather than to elementary analytic definitions.

What these cases have in common:

- They are uniquely specified by clear rules or properties,
- But there is no Turing machine that, given n, can output their n-th digit (or an approximation with guaranteed error bounds).

In other words, definability outstrips computability. This boundary region is philosophically important: it shows that the space of "mathematically meaningful" numbers is larger than the space of numbers we can generate algorithmically. π sits safely on the computable side, but the frontier is not far away once we start encoding general decision problems into the digits of a number.

8.4 π as a pedagogical gateway into deeper computability phenomena

Why focus on π at all, then, if the landscape is so vast? Precisely because π is familiar, rich, and safe, yet already contains the essential ideas:

- It is infinite and non-repeating, so the naive idea of "just storing all its digits" is clearly impossible.
- It is computable, so we can see concretely how a finite algorithm can capture an infinite structure.
- It has multiple independent definitions (geometric, analytic, algorithmic), illustrating how the same real can be characterized in many different ways.

Using π as a gateway, we can:

1. Introduce the concept of computable reals in a setting that feels natural (circle geometry, series, limits).
2. Transition smoothly to the idea of Kolmogorov complexity, showing how a short program can compress a long digit sequence.
3. Generalize to other classical constants (e, √2, ζ(s), etc.) and see that computability is widespread in the numbers we actually use.
4. Then push beyond: show that there are reals like Ω and diagonal constructions that are well-defined but non-computable, revealing the limits of algorithmic knowledge.

In this pedagogical arc, π serves as:

- A bridge between school-level ideas about numbers and the deep structure of computability theory,
- A counterexample to the naive belief that "every well-defined number must be computable,"
- A model for understanding how finite symbolic systems (programs, proofs) interface with infinite mathematical objects.

By understanding exactly how π can be determined by a computer without stored constants, we gain not just a new respect for this classical constant, but also a clearer view of the entire algorithmic landscape in which π lives—bounded on one side by well-behaved computable structure, and on the other by the vast, dark sea of the non-computable.

9. Conclusion

We began with a seemingly simple question: *Can π be determined algorithmically by a computer program without any stored constants?* Along the way, that question unfolded into a whole slice of computability theory and philosophy of mathematics. The answer, in its technical form, is "yes"—but everything interesting lives in what that "yes" actually means.

This conclusion gathers the threads: π as a paradigmatic computable real, finite programs as genuine interfaces to infinite structure, the epistemic content of being able to compute π, and some open questions that remain at the intersection of complexity, randomness, and ontology.

9.1 Summary: π as a paradigmatic computable real

π is an infinite, non-terminating, non-repeating decimal expansion; no finite list of digits can capture it. Yet π is also one of the best examples of a computable real: there exists a finite, explicit algorithm that, given any requested precision n, returns a rational approximation within error at most 10^{-n}(or 2^{-n}).

We saw multiple ways this can be done without "cheating" by storing π ahead of time:

- Series-based methods (Leibniz, Machin-like, Chudnovsky-type) that express π as an infinite sum of rational expressions with provably controllable tails.
- Geometric/Archimedean methods that bound π between perimeters of inscribed and circumscribed polygons, refined by iterative doubling and constructive square-root computations.
- Digit-extraction and spigot algorithms (e.g., BBP-type formulas) that allow generation of π's digits with purely arithmetic operations, sometimes even with local access to digit positions.

All of these can be implemented using only integers, rationals, arithmetic, and control flow. No π-digits are stored; π appears as a limit of operations, not as a preloaded constant.

From the standpoint of computability theory, this suffices: π is computable, and its digit sequence is fully determined by a finite program. In that precise, technical sense, a computer *can* determine π without stored constants.

9.2 Finite algorithms as windows onto infinite structures

A central conceptual move was to replace the image of "having all the digits of π" with the notion of an effective method:

- Not: a completed infinite list,
- But: a finite algorithm that can generate any finite prefix upon request.

Kolmogorov complexity framed this perspective in terms of compression:

- The shortest program that outputs the first $n$digits of π is much shorter than n.
- In fact, for a fixed π-algorithm P, we get roughly
  $K(\pi \upharpoonright n) \leq | P | + O(\log n)$,
  where $K(\cdot)$is Kolmogorov complexity and $\pi \upharpoonright n$is the first $n$digits.

In other words, a small program plus a description of how many digits we want serves as a *compressed description* of an arbitrarily long prefix of π. The program is a window onto an infinite structure: finite on the inside, infinite in what it can unfold.

This idea generalizes: any computable real is one whose infinite expansion can be "seen" through such a window. What makes π special is not only that it is computable, but that it is *deeply integrated* into geometry, analysis, and physics. Its computable status explains why we can work with it so effectively in practice.

9.3 The epistemic meaning of "being able to compute π"

What does it *epistemically* mean to say we can compute π?

Several layers emerged:

1. Operational knowledge
    - We possess concrete algorithms: given n, we know *how* to get an approximation with a proven error bound.
    - This "know-how" is what allows us to apply π in engineering, physics, and numerical computation.
2. Descriptive knowledge
    - We can summarize π compactly using formulas, series, and algorithms.
    - That compression, captured by Kolmogorov complexity, tells us that π's structure is not arbitrary; it is governed by simple laws.
3. Boundary knowledge
    - We understand that π belongs to a very small class of real numbers (the computable ones), which is countable and of measure zero in the continuum.
    - To "know" π is, therefore, to have access to an exceptionally structured point in an otherwise mostly uncomputable space.
4. Epistemic humility
    - While we can approximate π to billions of digits, we never possess all its digits as a completed object.
    - What we have—and all we can have as finite agents—is the ability to *generate* more information about π on demand, with guarantees.

From this viewpoint, “being able to compute π” is not trivial. It is a declaration that π is fully within the domain of effective mathematical knowledge: everything essential about π can, in principle, be encoded in proofs, algorithms, and finite processes. That stands in sharp contrast to non-computable reals, which may be definable but which forever resist this kind of effective grasp.

9.4 Open questions: complexity, randomness, and the ontology of number

Even with π as a well-understood example, several deeper questions remain open or at least philosophically unsettled.

1. Complexity of specific algorithms for π
    - We know that π is computable, but how close are our existing algorithms to optimal in terms of time and space as functions of the digit count n?
    - Are there algorithms that asymptotically dominate current methods, or are we near fundamental lower bounds?
    - How does the choice of base (e.g., binary vs hexadecimal vs decimal) affect the most efficient computational schemes?
2. Randomness properties of π’s digits
    - It is widely conjectured that π is *normal* in many bases, meaning its digits are statistically uniformly distributed.
    - Proving this remains an open problem.
    - If and when normality is established, π will provide a canonical example of a sequence that is:
        - Statistically very “random-looking,”
        - Yet algorithmically highly compressible and fully computable.
    - This would sharpen the conceptual distinction between statistical randomness and algorithmic randomness.
3. Boundary between definability and computability
    - Constants like Chaitin’s Ω show that there are reals whose definitions are deeply tied to computation, yet which are themselves non-computable.

- Where exactly do we draw the line between “legitimate” mathematical objects and those that are too entangled with undecidability to be useful?
- Is there a principled way, beyond taste and utility, to rank mathematical constants by epistemic accessibility?

4. Ontology of number: what “exists,” and in what sense?
    - A Platonist may say that π and non-computable reals alike exist in a timeless mathematical universe; computability simply marks which ones we can access.
    - A strict formalist may instead treat “existence” as a property relative to axiomatic systems and constructive procedures, giving computable numbers a privileged ontological status.
    - Intermediate positions see computability as articulating the *human* and *machine* boundary: the frontier between what finite agents can systematically know and what remains merely symbolically indicated.

In each of these debates, π functions as an anchor: a familiar constant whose status is clear enough to serve as a reference point, yet rich enough to be philosophically instructive.

In closing, the original question—whether π can be determined algorithmically without stored constants—turns out to be a gateway into a much larger story. π is indeed determinable in that sense: a finite program, built from nothing more exotic than integers, rationals, arithmetic, and control flow, suffices to unfold its infinite structure to arbitrary precision.

But in affirming this, we also confront the fact that π is *exceptional*: a rare island of computable order in a sea of non-computable possibilities. Finite algorithms, like the ones that compute π, are our windows onto the infinite. They show us not only how much can be known by finite means, but also, by contrast, how vast the uncomputable remainder must be.

References

[1] Turing, A. M. (1936). *On computable numbers, with an application to the Entscheidungsproblem.*
Proceedings of the London Mathematical Society, 2(42), 230–265.

[2] Church, A. (1936). *An unsolvable problem of elementary number theory.*
American Journal of Mathematics, 58(2), 345–363.

[3] Rogers, H. Jr. (1987). *Theory of Recursive Functions and Effective Computability.*
MIT Press.

[4] Pour-El, M. B., & Richards, J. I. (1989). *Computability in Analysis and Physics.*
Springer.

[5] Weihrauch, K. (2000). *Computable Analysis: An Introduction.*
Springer.

[6] Borwein, J. M., & Borwein, P. B. (1987). *Pi and the AGM: A Study in Analytic Number Theory and Computational Complexity.*
Wiley.

[7] Niven, I. (1956). *Irrational Numbers.*
Carus Mathematical Monographs, Mathematical Association of America.

[8] Chudnovsky, D. V., & Chudnovsky, G. V. (1989). *Approximations and complex multiplication according to Ramanujan.*
In *Ramanujan Revisited* (pp. 375–472). Academic Press.

[9] Bailey, D. H., Borwein, P. B., & Plouffe, S. (1997). *On the rapid computation of various polylogarithmic constants.*
Mathematics of Computation, 66(218), 903–913. (Introduces the BBP-type formula for π.)

[10] Archimedes of Syracuse. (ca. 3rd century BCE). *Measurement of a Circle.*
(Various modern translations; see any standard collection of Archimedes' works.)

[11] Cantor, G. (1874). *Über eine Eigenschaft des Inbegriffes aller reellen algebraischen Zahlen.*
Journal für die reine und angewandte Mathematik, 77, 258–262.

[12] Cantor, G. (1891). *Über eine elementare Frage der Mannigfaltigkeitslehre.*
Jahresbericht der Deutschen Mathematiker-Vereinigung, 1, 75–78.

[13] Li, M., & Vitányi, P. M. B. (3rd ed., 2008). *An Introduction to Kolmogorov Complexity and Its Applications.*
Springer.

[14] Chaitin, G. J. (1975). *A theory of program size formally identical to information theory.* Journal of the ACM, 22(3), 329–340.

[15] Chaitin, G. J. (1987). *Algorithmic Information Theory.* Cambridge University Press.

[16] Martin-Löf, P. (1966). *The definition of random sequences.* Information and Control, 9(6), 602–619.

[17] Downey, R. G., & Hirschfeldt, D. R. (2010). *Algorithmic Randomness and Complexity.* Springer.

[18] Calude, C. S. (2002). *Information and Randomness: An Algorithmic Perspective* (2nd ed.). Springer.

[19] Soare, R. I. (1987). *Recursively Enumerable Sets and Degrees.* Springer.

[20] Ambos-Spies, K., & Kucera, A. (1999). *Randomness in computability theory.* In *Computability Theory and Its Applications* (pp. 1–14). American Mathematical Society.

[21] Odifreddi, P. (1989). *Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers.* North-Holland.

These references collectively cover:

- Foundations of computability (Turing, Church, Rogers, Soare, Odifreddi, Weihrauch, Pour-El & Richards),
- Computable analysis and π algorithms (Borwein & Borwein, Niven, Chudnovsky & Chudnovsky, Bailey–Borwein–Plouffe, Archimedes),
- Cardinality and diagonalization (Cantor),
- Kolmogorov complexity and algorithmic randomness (Li & Vitányi, Chaitin, Martin-Löf, Downey & Hirschfeldt, Calude, Ambos-Spies & Kucera).

The author has published over 4,500 papers at https://www.researchgate.net/profile/Douglas-Youvan/research . Google Scholar has indexed these preprints, and if you want a particular reference, the AI mode of a Google search (Gemini) has efficiently catalogued these preprints.